Translator Output Specification

by

Dick Stevens

January 25, 2002

0. Introduction

This document describes the C++ code to be generated by the translator (Petor). The specification defines a set of functions that prescribe how the C++ code is to be generated.. Arguments of functions will be in terms of the EBNF specification of the Graph State File (GSF), dated August 17, 1999, together with calls to other functions.

The translator will generate a *.h file that gives the full definition of the graph class, together with the bodies of all methods.

Before specifying the functions, we give an outline of the organization of the C++ file to be generated. The *.h file has the following organization:

1) guard
2) banner
3) #includes
    a) GCL_Internal.h
    b) *.h files for included graphs
    c) *.h files for user defined data types
4) user-defined ordinary transition classes
    a) class definition
        i) data declarations
        ii) method prototypes
    b) method bodies
5) graph class definition
    a) data declarations
    b) graph constructor prototype
    c) graph destructor prototype
6) body of graph constructor
    a) construction of handles and objects
    b) association of graph ports with ports of internal graph objects
    c) connections of pairs of ports of internal graph objects
    d) function call to finalize construction
7) body of graph destructor
8) end guard

1. Primary Functions

The GSF does not specify the difference between a graph and an included graph. However there are some differences between the code to be generated for the two. In particular, there are two different base types GCL_MainGraph and GCL_InclGraph respectively. Moreover, item 6.d in the outline above is generated only for a main graph and not for an included graph. These are differences in the code to be generated.

There is also a requirement for the GSF itself: If a GSF specifies the association of a graph port with an internal transition port, then only an included graph my be generated. On the other hand, if a GSF specifies all of its graph ports to be associated with internal place ports, then either a main graph or an included graph may be generated.

In the following function definitions, we use the following notation:

function_name **(1: arg_1, arg_2, … :1)** ::= function_body

The function arguments are bounded by parentheses together with numbers and colons. The numbers indicate the level of nesting. Every left parenthesis at a specified level is balanced by a right parenthesis at the same level.

The function body specifies code to be evaluated, which will result in a value or specify code to be generated. The variables in the parentheses will appear in the function body. A function body may call other functions. When a function is called, each argument will be a call to another function, or an item in the GSF of the graph. Each item in the GSF will be uniquely identified using the full path with the item names separated by dots. The following fonts will be used:

- Formal specification is in times 14.
- Comments are in times 12.
- Function names are ***bold italics***.
- Formal arguments of functions are **bold**.
- GSF items are in times 14.

One final note before we begin: Most functions are intended to generate code. Some functions are intended solely for the purpose of calculating a value to be returned to the calling program. We make explicit the value to be returned by calling the following function, which evaluates its argument and returns the value.

return **(1: value :1)** ::= **value**

The following function is to evaluate each argument and print the values in the same order that they appear in the argument list..

print **(1: arg1, arg2, ... :1)** ::= self-explanatory.

At times we will want to indicate that a new line should begin with indenting.

newline **(1: n :1)** ::= a new line to begin, with the following line indented by n spaces.

text **(1:** literalString **:1)** ::= literalString

For example, ***text*** **(1:** hello **:1)** evaluates to the literal string "hello" (quotation marks not included).

cat **(1: arg_1, arg_2, … :1)** ::= **arg_1arg_2…**

The intent is to evaluate each of the arguments and to concatenate the results without separation by white space.

For example, ***cat*** **(1:** ***text*** **(2:** good **:2),** ***text*** **(2:** bye **:2) :1)** evaluates to the string "goodbye". Unless this function is called explicitly, successive elements of generated code are separated by white space. One common use for this is to generate code for a literal string contained in quotation marks. We define

quote **(1: name :1)**::=
    cat **(1:** ***text*** **(2: " :2), name,** ***text*** **(2: " :2) :1)**

For example, if **name** has the value DOG, then

quote **(1: name :1)** results in "DOG" (quotation marks included).

This function will be used to indicate when a new line is required. New lines and indentation may be entered at the discretion of the translator for ease of interpretation by a human reader. Generally, we do not use the function to indicate pretty printing.

if **(1: predicate :1)** ***then*** **(1: arg_1 :1)** ***else*** **(1: arg_2 :1)** ::= (self-explanatory)

This is a sequence of three functions. The intent is to evaluate the predicate. If it is true, then evaluate arg_1, else evaluate arg_2. If the else clause is omitted and the predicate is false, then the result is empty. We may also at times use the following construct, which is also self-explanatory:

if **(1: predicate_1 :1)** ***then*** **(1: arg_1 :1)**
    else if **(1: predicate_2 :1)** ***then*** **(1: arg_2 :1)**
    else if **(1: predicate_3 :1)** ***then*** **(1: arg_3 :1)**
    …
    else **(1: arg_n :1)** ::= (self-explanatory)

predicate **(1: expression :1)** ::= true iff the expression evaluates to true

The above function is usually not necessary. However, we will use it on occasion, when confusion might otherwise occur.

The graph state file specifies lists of items. For example, **Pgsf** contains a list of any number of things, each of which is one of the following: **NodeProto**, **Instance**, or **Arc**. For our purposes, it will be necessary to refer to all of each kind in a list. To do so, we will append ***list*** to the field name. For example, **Pgsf**.NodeProto_list is a list comprising every NodeProto in the **Pgsf**.

We will want to generate code corresponding to each item in a list. To do so, we specify the following functions, which are modeled after the corresponding LISP functions:

***car* (1: list :1)** ::= first element in the list

***cdr* (1: list :1)** ::= tail of the list (i.e., with first element removed)

***cons* (1: item, list :1)** ::= new list with the item as first element and given list as tail

***list* (1: item_1, item_2, … :1)** ::= new list with the numbered items

null ::= the empty list

Thus, for example, we will use ***car* (1: Pgsf**.NodeProto_list **:1)** to denote the first NodeProto in the NodeProto_list.

Using these functions we may define some functions below to be recursive.

In some cases the GSF grammar specifies multiple occurrences of items of the same type without providing a way to distinguish them. For example, in the *Range*, there are two items called *Expr*. In such cases, we use a subscript to distinguish them, as in *Expr.1* and *Expr.2*.

To identify items in the GSF grammar, we preferentially use the formal item name, like *Pgsf* and *Banner*. Sometimes no formal item name is present, but there is a keyword, like in the *Banner*, there are keywords *filename, author*, …, to identify the various items that are all strings. In that case, we use the keyword to identify the item.

For example, Pgsf denotes the entire graph state file. Pgsf.Banner.filename denotes the string following the keyword “filename” in the banner of the graph state file.

Further, the EBNF for the GSF uses symbols ?, +, and *, to indicate item occurrence as follows:

- ? to indicate 0 or 1 occurrences
- * to indicate 0 or more occurrences
- + to indicate 1 or more occurrences

We will treat these like lists and use the above LISP-like functions to deal with them. For example, we will use the following call to express a predicate that tests whether a list is empty:

```
if (1: list = null :1) then (1: … :1)
```

Sometimes we will want to generate code for each element in a list. We have the following function:

```
forEach (1: list, function :1)    :: =
    if (1: list = null :1) then (1: return (2: null :2) :1)
    else (1: function (2: car (3: list :3) :2)
        forEach (2: cdr (3: list :3), function :2) :1)
```

This formalizes the idea that the function specified by the function name should be applied to every element in the list.

Sometimes we will want to insert a separator between the successive elements of a list. Formally, we have the following function, which overloads the previous, by adding the separator as a third argument:

```
forEach  (1: list, function, separator :1)      :: =
    if (1: list = null :1) then (1: return (2: null :2) :1)
    else if (1: cdr (2: list :2) = null :1)
        then (1: function (2: car (3: list :3) :2) :1)
    else (1: function (2: car (3: list :3) :2)
        print (2: separator :2)
        forEach (2: cdr (3: list :3), function :2) :1)
```

We will want to detect whether an item is in a list. For that we define

```
isInList (1: item, list :1)  ::=
    if (1: list = null :1) then (1: return (2: false :2) :1)
    else if (1: item = car (2: list :2) :1) then (1: return (2: true :2) :1)
    else (1: isInList (2: item, cdr (3: list :3) :2) :1)
```

If no function is provided in the second argument, then the element itself is the desired value.

For example, if list denotes the 3-element list { 3.14, 2.72, 5.13 }, and we define the function

```
times2 (1: x :1)      ::=
    print (1: 2 * x :1)
```

i.e., to multiply by 2 and print the result.

Then the following sequence of function calls

```
    print (1: text (2: [ [  :2) :1)
    forEach (1: list, times2, text (2: ;  :2) :1)
    print (1: text (2: ] ]  :2) :1)
```

results in the following printed output:

```
    [ [ 6.28 ; 5.44 ; 10.26 ] ]
```

At times we will invoke a function that has more than one argument and provide all but one of the arguments. In that case, the missing argument is taken to be each element of the list. For example, consider the above list of 3 numbers, and define the function

times **(1: x, y :1)** ::=
    print **(1: x * y :1)**

i.e., to multiply x and y.

Then the following sequence of function calls

> ***print*** **(1:** ***text*** **(2:** { **:2) :1)**
> ***forEach*** **(1:** list**,** ***times*** **(2:** , 3 **:2),** ***text*** **(2:** , **:2) :1)**
> ***print*** **(1:** ***text*** **(2:** } **:2) :1)**

results in the multiplication of each element by 3, in the following notation:

    { 9.42 , 8.16 , 15.39 }

Where the second argument is the print function, we give the example:

> ***forEach*** **(1:** list**,** ***print*** **:1)**

which results in the following printed output:

    3.14 2.72 5.13

Finally, where the second argument is missing, we give the example:

> ***forEach*** **(1:** list**, :1)**

which evaluates each element of the list and returns a list of the resulting values.

Sometimes we will want to construct a list by concatenation of several given lists:

catList **(1: list_1, list_2, … list_n :1)** ::= self-explanatory

Sometimes we will want to know the number of elements in a list.

listSize **(1: List :1)** ::=
    if **(1: List** = **null :1)** ***then*** **(1:** ***return*** **(2: 0 :2) :1)**
    else **(1:** ***return*** **(:2** 1 + ***listSize*** **(3:** ***cdr*** **(4: List :4) :3) :2) :1)**

The following must be defined before translation begins.
graphType ::= ***text*** **(1:** MainGraph **:1)** or ***text*** **(1:** InclGraph **:1)**

We assume that **Pgsf** refers to the graph state file in the format specified in EBNF in the1-page document titled "GSF in the EBNF notation". We will occasionally refer to **Pgsf** in the definition of a function without specifically mentioning listing it as an argument. When we need to refer to the graph state file of an included graph, we will do so by using the following function:

***getPgsf* (1: FileName :1)**::= the graph state file in the file whose path is **FileName**.

The purpose for doing so will be to obtain information about the prototype of an included graph, whose graph state file is defined separately.

We organize the functions into sections according to their purpose. The next section defines the high level functions that specify the general organization of the C++ source code. This will be a single header file whose name is the same as the graph state file, except that where ".gsf" is the extension of the graph state file, ".h" is the extension of the generated file containing the C++ code. Following that will be other sections that define lower-level supporting functions.

2. Functions to define the output of translator from a given Graph State File

The first function we define here converts a file name with extension .gsf to the same file name with extension .h. The notation is unusual in that the formal argument invokes a function call. The intention is to recognize the argument as being a string that ends in `.gsf` and return a string with the `.gsf` ending replaced by the ending `.h`.

***gsf2h* (1: *cat* (2: filename, *text* (3:** .gsf **:3) :2) :1)** ::=
    ***cat* (1: filename, *text* (2:** .h **:2) :1)**

The following function defines the output of the translator. It calls a number of other supporting functions, which we define below in the document.

***C*++(1: graphType :1)** ::=
    ***Guard* (1: Pgsf**.Exterior.Prototype.ProtName.Name **:1)**
    ***C++Banner* (1: Pgsf**.Banner **:1)**
    ***Includes* (1: Pgsf :1)**
    ***TransitionClasses* (1: Pgsf :1)**
    ***GraphClass* (1: GraphType, Pgsf :1)**
    ***EndGuard* (1: Pgsf**.Exterior.Prototype.ProtName.Name **:1)**

The first three functions called above and the last one are "boiler-plate", and we define them in this section:

The Guard and EndGuard are the standard inclusion guards
that go at the beginning and end of *.h files

***Guard* (1: className :1)**::=
    ***print* (1: *text* (2:** #ifndef **:2), *cat* (2: className, *text* (3:** _h **:3) :2),**
        ***newline* (2:** 0 **:2),**

```
                text (2: #define  :2), cat (2: className, text (3: _h  :3) :2),
                newline (2: 0  :2) :1)

EndGuard (1: className :1)    ::=
        print (1: newline (2: 0  :2),
                text (2: #endif // :2), cat (2: className, text (3: _h  :3) :2),
                newline (2: 0  :2) :1)
```

The banner includes information derived from the banner in the GSF

```
C++Banner (1: banner :1)       ::=
        print (1: newline (2: 0 :2),
                text (2: /******************************  :2),
                newline (2: 0 :2),
                text (2: File name:  :2),           banner.filename,
                newline (2: 0 :2),
                text (2: Author: :2),               banner.author,
                newline (2: 0 :2),
                text (2: Revision:  :2),            banner.revision,
                newline (2: 0 :2),
                text (2: Purpose:  :2),             banner.purpose,
                newline (2: 0 :2),
                text (2: ****************************** /  :2)
                newline (2: 0 :2) :1)
```

Here we include the *.h files needed for the graph library, included graphs, and user-defined classes to be used as base types of tokens.

```
Includes (1: Pgsf :1)       ::=
        print (1: newline (2: 0 :2),
                text (2: #include "GCL/GCL_Internal.h"  :2), newline (2: 0 :2)
                text (2: #include "primitives/external.h" :2), newline (2: 0 :2) :1)
        forEach (1: Pgsf.Exterior.InclForm_list , includedGraphLine :1)
        forEach (1: Pgsf.Exterior.TypeForm_list, includedTypeLine :1)

includedGraphLine (1: InclForm :1) ::=
        print (1: text (2: #include  :2),
                gsf2h (2: InclForm.PathName :2),
                newLine (2: 0 :2) :1)
```

Note: The next function depends on the inclusion of the type form, which contains a list of the user-defined base-types, each with the path to its *.h file.

***includedTypeLine* (1: TypeForm :1)** ::=
    ***print* (1: *text* (2:** #include **:2),**
        TypeForm.PathName**,**
        ***newLine* (2:** 0 **:2) :1)**

3. Functions to make Transition Class Definitions

In this section we define the functions that define the transition class definitions. These functions support code generation for classes for user-defined ordinary transitions. A funny place is one which has a non-trivial family (i.e., a family with height ≥ 1) of input ports and/or a non-trivial family of output ports. We do not define special classes for funny places. Rather we construct descriptors for its inport and outport families and pass them as arguments to an overloaded version of the place constructor - one that takes two additional arguments.

Before proceeding, we give a brief discussion about a potential conflict of class names. Suppose the user defines a main graph G with two included graphs A and B. Suppose that included graph A defines a transition with class name Foo and that B also defines a transition with class name Foo. Then there is a conflict of names. Note that the Foo in A and the Foo in B may define transition classes that are either the same or different. In either case, there will be a problem in compiling and linking the main graph G.

One solution would be to nest the transition class definitions in the respective graph class definition, thus limiting the scope of the transition class to the graph or included graph in which it is used. Unfortunately this is not possible if the transition class is defined by a class template. In this situation, we are forced to define the transition class separately from the graph class, giving it the wider scope that we intended to avoid. This means our solution offers protection that is less than perfect.

Not wanting to pretend to offer protection that may fail under some circumstances, we make all node class definitions external to the graph class definitions.

Another question arises here: Suppose the user defines a transition that he wishes to use in several different graphs and included graphs. One imagines a library of useful transition classes. Our design of PGMT does not directly support such a library. However, it is quite possible for the user to define a number of included graphs. For example, it is possible to define an included graph that contains a single transition. In this way the user may assemble a number of included graphs in a directory and consider that this constitutes a library of included graphs. The only proviso here is that to avoid the name conflicts described above, all transitions in all included graphs in the library must have unique class names.

***TransitionClasses* (1: :1)** ::=
    ***forEach* (1: Pgsf**.NodeProto_list **,** ***singleTransitionClass* :1)**

The following function builds the class definition and methods for a given transition prototype. A class definition is generated only if the instance is an ordinary transition (i.e., not a place or special transition). For further information on the care and feeding of the special transitions in the GUI, see Appendix A.

```
singleTransitionClass (1: NodeProto :1)              ::=
        if (1: NodeProto_1 = text (2: transition :2) and
                NodeProto.Prototype.ProtName.Name != text (2: Pack :2) and
                NodeProto.Prototype.ProtName.Name != text (2: Unpack :2) :1)
        then (1: Guard (2: NodeProto.Prototype.ProtName.Name  :2)
                transitionClassDef (2: NodeProto :2)
                transitionConstructor (2: NodeProto :2)
                Destructor (2: NodeProto :2)
                transitionStatement (2: NodeProto :2)
                EndGuard (2: NodeProto.Prototype.ProtName.Name  :2) :1)

transitionClassDef (1: NodeProto :1)::=
        formalTemplateSpec (1: formalTemplateArgList
                (2: NodeProto :2) :1)
        transitionClass (1: NodeProto :1)

transitionClass (1: NodeProto :1)              ::=
        print (1: text (2: class :2), NodeProto.Prototype.ProtName.Name,
                text (2: : public GCL_OrdTran :2),
                text (2: { :2),
                newLine (2: 0 :2),
                text (2: public: :2) :1)
        ConstructorPrototype (1: text (2: transition :2),
                        NodeProto.Prototype :1)
        print (1: text (2: ; :2) :1)
        DestructorPrototype (1: NodeProto.Prototype :1)
        print (1: text (2: ; :2) :1)
        print (1: text (2: void :2) :1), TransitionStatementPrototype (1:  :1)
        print (1: text (2: ; :2),
                newLine (2: 0 :2),
                text (2: private: :2) :1)
        declarePorts (1: NodeProto :1)
        print (1: text (2: }; :2) :1)

TransitionStatementPrototype (1:  :1)              ::=
        print (1: text (2: tranStmt () :2) :1)

declarePorts (1: NodeProto :1)          ::=
        if (1: NodeProto_1 = text (2: transition  :2) :1)
        then (1: declareTranPorts (2: NodeProto.Prototype.Inports_list :2)
        declareTranPorts (2: NodeProto.Prototype.Outports_list :2) :1)

declareTranPorts (1: Ports_list :1)     ::=
    forEach (1: Ports_list, declareTokenPtr :1)
```

In the next function, the argument Port is either an inport or an outport.

```
declareTokenPtr (1: Port :1)     ::=
    print (1: text (2: GCL_WorkSpace_T < :2),
        Port.Mode.BaseType text (2: > :2),
        cat (2: text (3: _ :3), Port.Name :2), text (2: ; :2) :1)
```

The next function calls bodyDefs, which takes two arguments. The first specifies the graph name, and the second argument is each Node_Proto in the NodeProto_list.

```
transitionConstructor (1: NodeProto :1)      ::=
    classScopeSpec (1: NodeProto :1)
    ConstructorPrototype (1: text (2: transition  :2),
            NodeProto.Prototype  :1)
    transitionConstructorInitialization
        (1: NodeProto, catList (2: NodeProto.Prototype.Inports_list ,
                NodeProto.Prototype.Outports_list :2) :1)
    print (1: text (2: { :2),
        text (2: GCL_Descriptor * descriptor; :2),
        text (2: GCL_TranPort * tranport; :2) :1)
    transitionPorts (1: NodeProto,
            text (2: In :2), NodeProto.Prototype.Inports_list :1)
    transitionPorts (1: NodeProto,
            text (2: Out :2), NodeProto.Prototype.Outports_list :1)
    print (1: text (2: }  :2) :1)

classScopeSpec (1: Prototype :1)       ::=
    formalTemplateSpec (1: formalTemplateArgList (2: Prototype :2) :1)
    print (1: Prototype.ProtName.Name  :1)
    templateArgs (1: Prototype :1)
    print (1: text (2: :: :2) :1)

templateArgs (1: Prototype :1) ::=
    templateSpec (1: formalTemplateArgList (2: Prototype :2) :1)

templateSpec (1: TemplateArgList :1)        ::=
    if (1: TemplateArgList != null :1)
    then (1: print (2: text (3:  <  :3) :2)
            forEach (2: TemplateArgList, print, text (3: , :3) :2)
            print (2: text (3: > :3) :2) :1)

transitionConstructorInitialization (1: Prototype, PortsList :1)        ::=
    if (1: PortsList != null :1)
    then (1: print (2: text (3: : :3) :2)
            forEach (2: PortsList,
                    initializeTranPort (3: Prototype, :3), text (3: , :3) :2) :11)

initializeTranPort (1: Prototype, Port :1)    ::=
    print (1: cat (2: text (3: _ :3), Port.Name :2),
                    text (2: ( :2),
                    Port.Mode.TokHt,
                    text (2: + :2),
                    listSize (2: Port.Range_list :2),
                    text (2: , :2) :1)
    MW_DataType (1: Prototype, Port.Mode.BaseType :1)
    print (1: ) :1)

MW_DataType (1: Prototype, BaseType :1)         ::=
    if (1: isInList (2: BaseType,
                            formalTemplateArgList (3: Prototype :3) :2) :1)
    then (1: print (2: cat (3: BaseType , text (4: _Datatype :4) :3) :2) :1)
    else (1: print (2: text (3: Machine::getMW_Datatype ( :3),
                            quote (3: BaseType :3),
                            text (3: ) :3) :2) :1)
```

Note: Each node inport in the list of node inports is actually a family of inports. The same is true for the outports. A transition may have any number of families of inports and any number of families of outports. On the other hand, a place has just one family of inports and one family of outports. Moreover, the name of the inport family of a place is "INPUT", as prescribed by the PGM spec. Also, the name of the outport family of a place is "OUTPUT". We expect that the GUI will enforce this rule for places. Based on that assumption, we define the following functions for generating the node inports and node outports:

```
transitionPorts (1: Prototype, Direction, Ports_list :1)   ::=
    forEach (1: ports_list, makeTranPort (2: Prototype,
                    Direction, :2) :1)
```

```
makeTranPort (1: Prototype, Direction, Port :1)          ::=
        makeDescriptor (1: text (2: descriptor :2), Port.Range_list :1)
                print (1: text (2: tranport = new :2),
                        cat (2: text (3: GCL_Tran :3), Direction,
                                        text (3: port_T :3) :2),
                        text (2: < :2), Port.Mode.BaseType, text (2: > :2),
                        text (2: ( descriptor, :2),
                        cat (2: text (3: &_ :3), Port.Name :2), text (2: , :2),
                        quote (2: Port.Name :2), text (2: , :2) :1)
                        modeArgs (1: Prototype, Port.Mode :1)
                        print (1: text (2: ); :2),
                                text (2: _attachTranPort (tranport); :2) :1)

modeArgs (1: Prototype, Mode :1)     ::=
        print (1: Mode.TokHt , text (2: , :2) :1)
        MW_DataType (1: Prototype, Mode.BaseType :1)

Destructor (1: NodeProto :1)    ::=
        classScopeSpec (1: NodeProto :1)
        DestructorPrototype (1: NodeProto.Prototype :1)
        print (1: text (2: {} :2) :1)

TransitionStatement (1: NodeProto :1)                ::=
        formalTemplateSpec (1: formalTemplateArgList (2: Prototype :2) :1)
        print (1: text (2: void :2), Prototype.ProtName.Name :1)
        templateArgs (1: Prototype :1)
        print (1: text (2: :: :2) :1)
        TransitionStatementPrototype (1: NodeProto.Prototype :1)
        print (1: text (2: { :2) :1)
        header (1: NodeProto :1)
        print (1: Prototype.TrStmt :1)
        footer (1: NodeProto :1)
        print (1: text (2: } :2) :1)

header (1: NodeProto :1)          ::=
        forEach (1: NodeProto.Inports_list, declareInputVar :1)
        forEach (1: NodeProte.Outports_list, declareOutputVar :1)
```

Note: In three of the next four functions, a check is made to determine whether the TokHt = 0 for an **Inport** or **Outport**. This is impossible to determine if the TokHt is an expression. In that circumstance, the translator must assume that an expression is, by definition, not 0.

```
declareInputVar (1: Inport :1)          ::=
        if (1: Inport.Mode.TokHt = 0 and listSize (2: Inport.Range_list :2) = 0 :1)
        then (1: print (2: Inport.Mode.BaseType,
                                Inport.Name,
                                text (3: ; :3),
                                cat (3: text (4: _ :4), Inport.Name :3),
                                text (3: . getLeaf ( :3),
                                Inport.Name,
                                text (3: ); :3) :2) :1)
        else (1: print  (2: text (3: GCL_WorkSpace_T < :3),
                                Inport.Mode.BaseType,
                                text (3: > & :3),
                                Inport.Name,
                                text (3: = :3),
                                cat (3: text (4: _ :4), Inport.Name :3),
                                text (3: ; :3) :2) :1)

declareOutputVar (1: Outport :1)      ::=
        if (1: Outport.Mode.TokHt = 0 and
                        listSize (2: Outport.Range_list :2) = 0  :1)
        then (1: print (2: Outport.Mode.BaseType,
                                Outport.Name,
                                text (3: ; :3) :2) :1)
        else (1: print (2: text (3: GCL_WorkSpace_T < :3),
                                Inport.Mode.BaseType,
                                text (3: > & :3),
                                Outport.Name,
                                text (3: = :3),
                                cat (3: text (4: _ :4), Outport.Name :3),
                                text (3: ; :3) :2) :1)

footer (1: NodeProto :1) ::=
        forEach (1: NodeProto.Outports_list, defineOutputVar :1)

defineOutputVar (1: Outport :)          ::=
        if (1: Outport.Mode.TokHt = 0 and
                    listSize (2: Outport.Range_list :2) = 0 :1)
        then (1: print (2: cat (3: text (4: _ :4), Outport.Name :3),
                                text (3: . putLeaf ( :3),
                                Outport.Name,
                                text (3: ); :3) :2) :1)
```

4. Functions to build the graph class definition

This section defines the functions that create the graph class definition.

The first function in this section creates the graph class definition along with the two methods of the graph class: its constructor and destructor. The function that builds the graph constructor and its supporting functions are defined in later sections.

```
GraphClass (1: graphType :1) ::=
        GraphProto (1: graphType :1)
        GraphConstructor (1: graphType :1)
        destructor (1: Pgsf.Exterior.Prototype :1)
```

This is the prototype for the graph being defined here (recall that **graphType** was defined above):

```
GraphProto (1: graphType :1) ::=
        formalTemplateSpec (1:
                formalTemplateArgList (2: Pgsf.Exterior.Prototype :2) :1)
        GraphClassDefinition (1: graphType, Pgsf :1)
```

Here is the graph class definition

```
GraphClassDefinition (1: graphType, Pgsf :1)      ::=
        print (1: text (2: class :2),
                Pgsf.Exterior.Prototype.ProtName.Name ,
                text (2: : public :2),
                cat (2: text (3: GCL_ :3), graphType :2),
                newline (2: 0 :2), text (2: { :2),
                newline (2: 0 :2), text (2: public: :2),
                newline (2: 0 :2) :1)
        ConstructorPrototype (1: graphType, Pgsf.Exterior.Prototype :1),
        print (1: text (2: ; :2) :1)
        DestructorPrototype (1: Pgsf.Exterior.Prototype :1)
        print (1: text (2: ;  :2),
                newLine (2: 0 :2),
                text (2: }; :2) :1)
```

5. Functions to build the graph constructor and destructor

```
GraphConstructor (1: graphType :1) ::=
        classScopeSpec (1: Pgsf.Exterior.Prototype :1)
        ConstructorPrototype (1: graphType, Pgsf.Exterior.Prototype :1)
        if (1: graphType = text (2: MainGraph :2) :1)
        then (1: print (2:
                text (3: : GCL_MainGraph ( cpRanks, graphRanks, :3),
                text (3: portAssignment, maxLatency, :3) :2) :1)
        else (1: print (2: text (3: : GCL_InclGraph ( :3) :2) :1)
        print (1: quote (2: Pgsf.Exterior.Prototype.ProtName.Name :2),
                        text (2: , :2),
                quote (2: Pgsf.Banner.revision :2), text (2: , :2),
                quote (2: gsf2h (3: Pgsf.Banner.filename :3) :2), text (2: ) { :2) :1)
        graphPorts (1: GraphType, text (2: GCL_GraphInport_T :2),
                                Pgsf.Exterior.Prototype.Inports_list :1)
        graphPorts (1: GraphType, text (2: GCL_GraphOutport_T :2),
                                Pgsf.Exterior.Prototype.Outports_list :1)
        instances (1: Pgsf.Instance_list :1)
        portAssociation (1: Pgsf.Exterior.PortAssoc :1)
        arcs (1: Pgsf.Arc_list :1)
        conclude (1: graphType, Pgsf :1)
        print (1: text (2: } :2) :1)

graphPorts (1: GraphType, TypeName, Ports_list :1)   ::=
        forEach (1: Ports_list, makeGraphPort
                                (2: GraphType, TypeName, :2) :1)

makeGraphPort  (1: GraphType, TypeName, Port :1)   ::=
        makeDescriptor  (1: text (2: _descriptor :2), Port.Range_list :1)
        makeStringIDs  (1: Port.Name, (2: TypeName text  (3: < :3)
                                Port.Mode.BaseType text  (3: > :3) :2) :1)
        makeHandle (1: text (2: GRAPHPORT :2) :1)
        if (1: graphType = text (2: MainGraph :2) :1)
        then (1: makeGraphPortObjects  (2: TypeName, Port :2) :1)

makeGraphPortObjects  (1: TypeName, Port :1)  ::=
        print  (1: text (2: { :2),
                TypeName, text (2: < :2), Port.Mode.BaseType, text (2: > * :2),
                        Port.Name, text (2: ; :2),
                text (2: for (int j = 0; j < _descriptor -> getLeafCount(); j++) { :2),
                Port.Name, text (2: = new :2),
                TypeName, text (2: < :2), Port.Mode.BaseType, text (2: > ( :2) :1)
        modeArgs (1: Pgsf.Exterior.Prototype, Port.Mode :1)
        print (1: text (2: ); :2),
                text (2: _graphObjHdl -> addGraphObj ( (GCL_GraphObj *) :2),
                Port.Name,
                text (2: ); } } :2) :1)

conclude (1: graphType, Pgsf :1)      ::=
        if (1: graphType = text (2: MainGraph  :2) :1)
        then (1: print  (2: text (3: _initializeGraph(); :3) :2) :1)

Destructor (1: Prototype :1)     ::=
        classScopeSpec (1: Prototype :1)
        DestructorPrototype (1: Prototype :1)
        print (1: text (2: {} :2) :1)
```

<u>6. Functions to build handles and objects for the handles</u>

```
instances (1: Instance_list :1)           ::=
        forEach (1: Instance_list, makeInstance :1)

makeInstance (1: Instance :1)  ::=
        makeDescriptor (1: text (2: _descriptor :2), Instance.Range_list :1)
        makeStringIDs (1: Instance.label , makeClassName (2: Instance :2) :1)
        makeHandle (1: getObjType  (2: Instance :2) :1)
        makeObjects (1: Instance :1)

makeStringIDs (1: familyName, className :1)     ::=
        print (1: text (2: _stringIDs = new GCL_StringIDs ( :2),
                quote (2: familyName :2),
                text (2: , :1),
                quote (2: className :2),
                text (2: , :2),
                text (2: _graphClassName, :2),
                text (2: _graphVersion, :2),
                text (2: _graphFileName ); :2) :1)

makeHandle (1: objectType :1)         ::=
        print (1: text (2: _graphObjHdl = new GCL_GraphObjHdl ( :2),
                text (2: (GCL_Graph *) this, :2), objectType, text (2: , :2),
                text (2: _descriptor, :2),
                text (2: _stringIDs ); :2) :1)

getObjType (1: Instance :1)      ::=
        if (1: Instance.= = text (2: transition :2) :1)
        then (1: text (2: TRAN :2) :1)
        else if (1: Instance.= = text (2: place :2) :1)
        then (1: text (2: PLACE :2) :1)
        else (1: text (2: INCLGRAPH :2) :1)

makeObjects (1: Instance :1)             ::=
        print (1: text (2: { :2) :1)
        declareInstanceVars (1: Instance :1)
        if (1: Instance.= != text (2: place :2) :1)
        then (1: makeGipVars (2: Instance :2) :1)
        makeObjectsRecurs (1: Instance.Range_list, Instance :1)
        print (1: text (2: } :2) :1)

declareInstanceVars (1: Instance :1) ::=
        makeClassName (1: Instance :1)
        print (1: text (2: * :2), Instance.label, text (2: ; :2) :1)

makeClassName (1: Instance :1)       ::=
        if (1: Instance.= = text (2: place :2) :1)
        then (1: makePlaceClassName
                        (2: getPrototype (3: Instance :3), Instance :2) :1)
        else if (1: Instance.ProtCall.Name = text (2: Pack :2) :1)
        then (1: print (2: text (3: GCL_Pack_T < :3) :2),
                        Instance.FamInit.Mode_1.BaseType, text (2: > :2) :1)
        else if (1: Instance.ProtCall.Name = text (2: Unpack :2) :1)
        then (1: print (2: text (3: GCL_Unpack_T < :3) :2),
                        Instance.FamInit.Mode_1.BaseType, text (2: > :2) :1)
        else (1: print (2: Instance.ProtCall.Name :2),
                        bindTemplates (2: Instance :2) :1)
```

Note: In the next function, we assume that for a place, the Instance.ProtCall.Name is either "Queue" or "Gvar".

```
makePlaceClassName (1: NodeProto , Instance :1)        ::=
        if (1: NodeProto = null :1)
        then (1: print (2:
                cat (3: text (4: GCL_ :4),
                        Instance.ProtCall.Name, text (4: _T :4) :3) :2) :1)
        else if (1: NodeProto_1 = text (2: queue :2) :1)
        then (1: print (2: text (3: GCL_Queue_T :3) :2) :1)
        else (1: print (2: text (3: GCL_GVar_T  :3) :2) :1)
        print (1: text (2: < :2),
                        Instance.FamInit.Mode_1.BaseType, text (2: > :2) :3)

getBasetype (1: Mode :1)                    ::=
        Mode.BaseType

makeObjectsRecurs (1: Range_list, Instance :1)   ::=
        if (1: Range_list = null :1)
        then (1: makeSingleObject (2: Instance :2) :1)
        else (1: startLoop (2: car (3: Range_list :3)
                makeObjectsRecurs (3: cdr (4: Range_list :4), Instance :3)
                endLoop (3:  :3) :2) :1)
```

Note: The following function creates an instance for the handle. The code to be generated depends on what kind of instance it is transition, place, or included graph.

```
makeSingleObject (1: Instance :1)      ::=
        if (1: Instance.= = text (2: place :2) :1)
        then (1: makePlace (2: Instance :2) :1)
        else (1: makeNonPlace (2: Instance :2) :1)

makeNonPlace (1: Instance :1)                   ::=
        bindGipVars (1: Instance :1)
        print (1: Instance.label, text (2: = new :2) :1)
        makeClassName (1: Instance :1)
        print (1: text (2: ( :2) :1)
                argBindings (1: Instance :1)
        print (1: text (2: ); :2) :1)
        addToHandle (1: Instance :1)

bindGipVars (1: Instance :1)    ::=
        forEach (1: Instance.Bindings, assignValueToGip (2: Instance,
          findFormalGip (3: Binding.Name,
                   getPrototype (4: Instance :4) :3).Mode,  :2) :1)
```

Note: In the following function a check is made to determine whether the mode of a Gip has TokHt = 0. This is impossible to check if an expression is present. In the context of a GIP, the TokHt can only be specified meaningfully by an integer. However, if the GUI does not enforce this rule, the translator must assume that an expression is, by definition, not 0.

```
assignValueToGip (1: Instance, Mode, Binding :1)        ::=
        if (1: Mode.TokHt = 0 :1)
        then (1: print (2: Instance.Bindings.Name, text (3: = :3),
                       Instance.Bindings.Value.Leaf :2), text (2: ; :2) :1)
        else (1: makeToken (2: Instance.Bindings.Name,
                                    Instance.Bindings.Value, Mode :2) :1)
```

```
findFormalGip (1: Name, Gips_list :1)        ::=
        if (1: Gips_list = null :1) then (1: return (2: null :2) :1)
        else (1: if (2: Name = (3: car (4: Gips_list :4) :3).Name  :2)
                then (2: return (3: car (4: Gips_list :4) :3) :2)
                else (2: return (3: findFormalGip
                        (4: Name, cdr (5: Gips_list :5) :4) :3) :2) :1)
```

Note: The next function creates a queue or gvar. If it is a funny queue or gvar, then we create the descriptors for the inports and outports to pass to the place constructor. This requires us to look at the place prototype, which contains the ranges needed to build the descriptors. This function also initializes the place by building a single token. If the place is a gvar, then the height of the token is equal to the token height specified in the instance mode. If the place is a queue, then the height is one greater to account for their being multiple tokens in the queue.

```
makePlace (1: Instance :1)                ::=
        if (1: nonTrivialRanges (2: Instance :2) :1)
                then (1: makePortDescriptors (2: Instance :2) :1)
        if (1: Instance.Value != null :1)
        then (1: makeToken (2: text (3: genericToken :3), Instance.Value,
                           Instance.FamInit.Mode :2) :1)
        print (1: Instance.label, text (2: = new :2) :1)
        makeClassName (1: Instance :1)
        print (1: text (2: ( :2) :1)
        placeArgBindings (1: Pgsf.Exterior.Prototype , Instance :1)
        if (1: nonTrivialRanges (2: Instance :2) :1)
                then (1: print (2: text
                (3: , inportDescriptor, outportDescriptor :3) :2) :1)
        if (1: Instance.Value != null :1)
            then (1: print (2: (3: text (4: , genericToken :4) :3) :2) :1)
        print (1: text (2: ); :2) :1)
        addToHandle (1: Instance :1)

nonTrivialRanges (1: Instance :1)    ::=
        if (1: getPrototype (2: Instance :2) = null :1)
        then (1: return (2: false :2) :1)
        else if (1: getPrototype (2: Instance :2).Inports.Range_list != null :1)
        then (1: return (2: true :2) :1)
        else if (1: getPrototype (2: Instance :2).Outports.Range_list != null :1)
        then (1: return (2: true :2) :1)
        else (1: return (2: false :2) :1)

makePortDescriptors (1: Instance :1)::=
        declarePlaceGipVars (1: Instance :1)
        print (1: text (2: GCL_Descriptor * inportDescriptor; :2),
            text (2: GCL_Descriptor * outportDescriptor; :2) :1)
        makeDescriptor (1: text (2: inportDescriptor :2),
            getPrototype (2: Instance :2).Inports.Range_list  :1)
        makeDescriptor (1: text (2: outportDescriptor :2),
            getPrototype (2: Instance :2).Outports.Range_list :1)

declarePlaceGipVars (1: Instance :1)::=
        forEach (1: Instance.Bindings ,
            createGipVar (2: , getPrototype (3: Instance :3) :2) :1)
```

Note: In the following function a check is made to determine whether the mode of a Gip has TokHt = 0. This is impossible to check if an expression is present. In the context of a GIP, the TokHt can only be specified meaningfully by an integer. However, if the GUI does not enforce this rule, the translator must assume that an expression is, by definition, not 0.

```
createGipVar (1: Binding , Prototype :1)    ::=
        if (1: findGipMode
            (2: Binding.Name, Prototype.Gips_list  :2).TokHt = 0 :1)
        then (1: print (2:
            findGipMode (3: Binding.Name, Prototype.Gips_list :3).BaseType,
                Binding.Name, text (3: = :3), Binding.Value.Leaf,
                text (3: ; :3) :2) :1)
```

```
    else (1: print (2: text (3: GCL_Token_T < :3), (3: findGipMode
                (4: Binding.Name, Prototype.Gips_list :4) :3).BaseType,
            text (3: > :3), Binding.Name, text (3: ; :3) :2)
            makeToken (2: Binding.Name, Binding.Value,
                findGipMode (3: Binding.Name,
                        Prototype.Gips_list :3) :2) :1)

findGipMode (1: Name, Gips_list :1) ::=
    if (1: car (2: Gips_list :2).Name = Name :1)
    then (1: car (2: Gips_list :2).Mode :1)
    else (1: findGipMode (2: Name, cdr (3: Gips_list :3) :2) :1)
```

Note: The following function should produce just two values, because there should be just a single mode, which determines the token height and base type of its tokens.

```
placeArgBindings (1: Prototype, Instance :1)      ::=
    modeArgs (1: Prototype, Instance.FamInit.Mode_1 :1)

argBindings (1: Instance :1)    ::=
    forEach (1: catList (2: protCallBindings (3: Instance :3),
                            famInitBindings (3: Instance :3),
                            gipBindings (3: Instance :3) :2), , text (2: , :2) :1)

addToHandle (1: Instance :1)  ::=
    print (1:
        text (2: _graphObjHdl -> addGraphObj ( (GCL_GraphObj *)  :2) :1)
    Instance.label text (1: ); :1)
```

7. Functions to build constructor and destructor prototypes

```
ConstructorPrototype (1: kind, Prototype :1)               ::=
    print (1: Prototype.ProtName.Name, text (2: ( :2) :1)
    forEach (1: FormalArgList (2: kind, Prototype :2),
                print, text (2: , :2) :1)
    print (1: text (2: ) :2) :1)

FormalArgList  (1: kind, Prototype :1)       ::=
    catList  (1: ProtNameArgs  (2: Prototype.ProtName :2),
                 FamNameArgs  (2: Prototype.FamName :2),
                 GipArgs  (2: Prototype.Gips  :2),
                 PgmtArgs (2: kind :2) :1)
```

PgmtArgs are needed for all graph constructors that are derived from MainGraph vice those that are derived from InclGraph. See Joe's Pep interface document.

```
PgmtArgs (1: kind :1)     ::=
        if  (1: kind = text (2: MainGraph :2) :1)
        then (1: list (2: text (3: const CPRanks & cpRanks :3),
                text (3: const GraphRanks & graphRanks :3),
                text (3: const GraphPortAssignment  & portAssignment :3),
                text (3: double maxLatency = 1.e18 :3) :2) :1)
        else (1: null :)

ProtNameArgs (1: ProtName :1)       ::=
        forEach (1: ProtName.Template_list, classDeclare :1)

classDeclare (1: Name :1)               ::=
        print (1: text (2: GCL_BaseType :2),
                        cat (2: Name, text (3: _Datatype :3) :2) :1)

FamNameArgs (1: FamName :1)              ::=
        forEach (1: FamName.FormalArg_list, modeDeclare :1)

modeDeclare (1: FormalArg :1)            ::=
        print (1: text (2: unsigned int :2),
                FormalArg.Name_1,
                text (2: , GCL_BaseType :2),
                cat (2: FormalArg.Name_2, text (3: _Datatype :3) :2) :1)

GipArgs (1: Gips :1)             ::=
        forEach (1: Gips_list, DeclareGip :1)
```

Note: In the following function a check is made to determine whether the mode of a Gip has TokHt = 0. This is impossible to check if an expression is present. In the context of a GIP, the TokHt can only be specified meaningfully by an integer. However, if the GUI does not enforce this rule, the translator must assume that an expression is, by definition, not 0.

```
DeclareGip (1: Gip :1)    ::=
        if (1: Gip.Mode.TokHt = 0 :1)
        then (1: print (2: Gip.Mode.BaseType,  Gip.Name :2) :1)
        else (1: print (2: text (3: GCL_Token_T < :3),
                        Gip.Mode.BaseType,
                    text (3: > :3),
                    Gip.Name :3) :2) :1)

DestructorPrototype (1: Prototype :1)                 ::=
        print (1: cat (2: text (3: ~ :3), Prototype.ProtName.Name  :2),
                text (2: () :2) :1)
```

8. Functions to build a formal template argument list

This function builds the formal template declaration

```
formalTemplateSpec (1: TemplateArgList :1)       ::=
        if (1: TemplateArgList != null :1)
        then (1: print (2: text (3: template < :3) :2)
                forEach (2: TemplateArgList, formalTemplateArg,
                                        text (3: , :3) :2)
                print (2: text (3: > :3) :2) :1)
```

Concatenate the formal type names in the list of Templates in the ProtName with each second Name (which corresponds to the base type) in the list of FormalArgs in the FamName:

```
formalTemplateArgList (1: Prototype :1)    ::=
        catList (1: Prototype.ProtName.Template_list ,
                        Prototype.FamName.FormalArg_list.Name_2 :1)

formalTemplateArg (1: Name :1)               ::=
        print (1: text (2: class :2), Name :1)
```

9. Functions to bind template arguments

```
bindTemplates (1: Instance :1)           ::=
        templateBindingSpec (1: templateBindingList (2: Instance :2) :1)

templateBindingList (1: Instance :1)           ::=
        catList (1: Instance.ProtCall.BaseType_list,
                        forEach (2: Instance.FamInit.Mode_list, getBaseType :2) :1)

templateBindingSpec (1: TemplateBindingList :1)          ::=
        if (1: TemplateBindingList != null :1)
        then (1: print (2: text (3: < :3) :2)
                forEach (2: TemplateBindingList, print, text (3: , :3) :1)
                print (2: text (3: > :3) :2) :1)
```

10. Functions to bind constructor arguments

```
makeGipVars (1: Instance :1)  ::=
        forEach (1: getPrototype (2: Instance :2).Gips_list,
                        declareGipVar (2: Instance, :2) :1)

declareGipVar (1: Instance, Gips :1) ::=
        if (1: Gips.Mode.TokHt = 0 :1)
        then (1: print (2: Gips.Mode.BaseType, Gips.Name, text (3: ; :3) :2) :1)
        else (1: print (2: text (3: GCL_Token_T < :3),
                Gips.Mode.BaseType, text (3: > :3), Gips.Name,
                        text (3: ; :3) :2) :1)
```

The next function finds the prototype (transition, place, or included graph) with the given class name. If it is a transition or place, then the desired prototype is somewhere in the current Pgsf. If it is an included graph, then the desired prototype is in the **Pgsf.**Exterior of that graph state file whose name is ***cat*** **(: Name,** ***text*** **(:** .h **:) :)**. That graph state file is found in the **Pgsf.**Exterior.InclGraph form of the current **Pgsf**. While all other functions provide arguments containing all information needed for the function, this one does not explicitly list the current **Pgsf**. Nonetheless, it is assumed to be available for this function's execution.

```
getPrototype (1: Instance :1)    ::=
        if (1: Instance.= = text (2: inclgraph :2) :1)
        then (1: getInclProto (2: Instance.ProtCall.Name,
                                Pgsf.Exterior.InclForm.NamePair_list  :2) :1)
        else (1: getNodeProto (2: Instance.ProtCall.Name,
                                Pgsf.NodeProto_list :2) :1)

getInclProto (1: Name, NamePairList :1)   ::=
        if (1: NamePairList = null :1)
        then (1: return (2: null :2) :1)
        else if (1: car (2: NamePairlist :2).Name_1 = Name :1)
        then (1: return (2: getExternalProto (3: car
                        (4: NamePairList :4).Name_2 :3) :2) :1)
        else (1: return (2: getInclProto
                        (3: Name, cdr (4: InclForm.N :4) :3) :2) :1)

getExternalProto (1: FileName :1)    ::=
        return (1: FileName.Pgsf.Exterior.Prototype :)
```

```
getNodeProto (1: Name, NodeProto_list :1)          ::=
        if (1: NodeProto_list = null :1)
        then (1: return (2: null :2) :1)
        else if (1: car (2: NodeProto_list :2).Prototype.ProtName.Name
                        = Name :1)
        then (1: return (2: car (3: NodeProto_list :3).Prototype :2) :1)
        else (1: return (2: getNodeProto
                        (3: Name, cdr (4: NodeProto_list :4) :3) :2) :1)

declareInitVar (1: Instance, Binding :1)      ::=
        print (1: text (2: GCL_Token_T < :2) :1)
        getGipMode (1: Instance.Gips_list, Binding.Name :1).BaseType
        print (1: text (2: > :2),
                Binding.Name,
                text (2: ; :2) :1)

getGipMode (1: FormalGips_list, Name :1)                  ::=
        if (1: FormalGips_list = null :1)
        then (1: return (2: null :2) :1)
        else if (1: car (2: FormalGips_list :2).Name = Name :1)
        then (1: return (2: car (3: FormalGips_list :3).Mode :2) :1)
        else (1: return (2: getGipMode
                        (3: cdr (4: FormalGips_list :4), Name :3) :2) :1)

protCallBindings (1: Instance :1)       ::=
        forEach (1: Instance.ProtCall.BaseType_list,
                        MW_Datatype (2: getPrototype (3: Instance :3), :2) :1)

famInitBindings (1: Instance :1)        ::=
        forEach (1: Instance.famInit.Mode_list,
                        modeArgs (2: getPrototype (3: Instance :3), :2) :1)

gipBindings (1: Instance :1)     ::=
        forEach (1: getPrototype (2: Instance :2).Gips_list,
                        getFormalGipName (2:  :2) :1)

getFormalGipName (1: Gip :1)                ::=
        return (1: Gip.Name :1)
```

11. Functions to associate a graph port with an internal node port

```
portAssociation (1: PortAssoc :1)      ::=
        inPortAssoc (1: PortAssoc.input_list :1)
        outPortAssoc (1: PortAssoc.output_list :1)

inPortAssoc (1: Input_list :1)   ::=
        forEach (1: Input_list, associateInport :1)

associateInport (1: Input :1)    ::=
        print (1: text (2: _associateGraphInport ( :2),
                quote (2: Input.Name_1 :2), text (2: , :2),
                quote (2: Input.label :2), text (2: , :2),
                quote (2: Input.Name_2 :2), text (2: ); :2) :1)

outPortAssoc (1: Output_list :1)       ::=
        forEach (1: Output_list, associateOutport :1)

associateOutport (1: Output :1)        ::=
        print (1: text (2: _associateGraphOutport ( :2),
                quote (2: Output.label :2), text (2: , :2),
                quote (2: Output.Name_2 :2), text (2: , :2),
                quote (2: Output.Name_1 :2), text (2: ); :2) :1)
```

12. Functions to connect ports of two graph objects

```
arcs (1: Arc_list :1)              ::=
        forEach (1: Arc_list, doArc :1)
```

Note: in the next function, the last argument of *makeConnections* is the "when" expression.

```
doArc (1: Arc :1)  ::=
        sizeIndexVectors (1: Arc.Connect_1, Arc.Connect_2 :1)
        print (1: text (2: { :2) :1)
        makeConnections (1: Arc.Range_list, Arc.Connect_1,
                                Arc.Connect_2, Arc.WhenExpr :1)
        print (1: text (2: } :2) :1)

sizeIndexVectors (1: Connect_1, Connect_2 :1)    ::=
        print (1: text (2: _portIndexResize ( :2),
                listSize (2: Connect_1.label.Expr :2), text (2: , :2),
                listSize (2: Connect_1.Name.Expr :2), text (2: , :2),
                listSize (2: Connect_2.label.Expr :2), text (2: , :2),
                listSize (2: Connect_2.Name.Expr :2), text (2: ); :2) :1)
```

The next function is recursively defined to generate a nested loop

```
makeConnections (1: Range_list, Connect_1, Connect_2, When :1) ::=
    if (1: Range = null :1)
    then (1: makeSingleConnection
                (2: Connect_1, Connect_2, When :2) :1)
    else (1: startLoop (2: car (3: Range_list :3) :2)
        makeConnections (2: cdr (3: Range_list :3),
                                        Connect_1, Connect_2, When :2)
        endLoop (2:  :2) :1)

startLoop (1: singleRange :1)   ::=
    print (1: text (2: for ( int :2), singleRange.Name, text (2: = :2),
        singleRange.Expr_1, text (2: ; :2),
        singleRange.Name, text (2: <= :2),
            singleRange.Expr_2, text (2: ; :2),
        singleRange.Name, text (2: ++) { :2) :1)

endLoop (1:  :1)    ::=
    print (1: text (2: } :2) :1)

makeSingleConnection (1: Connect_1, Connect_2, When :1)  ::=
    defineVec (1: text (2: _fromObjIndices :2),
                Connect_1.label.Expr_list, 0 :1)
    defineVec (1: text (2: _fromPortIndices :2),
                Connect_1.Name.Expr_list, 0 :1)
    defineVec (1: text (2: _toObjIndices :2),
                Connect_2.label.Expr_list, 0 :1)
    defineVec (1: text (2: _toPortIndices :2),
                Connect_2.Name.Expr_list, 0 :1)
    if (1: When != null :1) then (1: text (2: if (  :2) When text (2: ) :2) :1)
    print (1: text (2: _connectPorts ( :2),
        quote (2: Connect_1.label :2), text (2: , _fromObjIndices, :2),
        quote (2: Connect_1.Name :2), text (2: , _fromPortIndices, :2),
        quote (2: Connect_2.label :2), text (2: , _toObjIndices, :2),
        quote (2: Connect_2.Name :2), text (2: , _toPortIndices ); :2) :1)
```

The following function defines the elements of the index vector

```
defineVec (1: VectorName, ExprList, n :1) ::=
        if (1: ExprList != null :1)
        then (1: print (2: VectorName,
                        text (3: [ :3),
                        n,
                        text (3: ] = :3),
                        car (3: ExprList :3),
                        text (3: ; :3) :2)
                defineVec (2: VectorName, cdr (3: ExprList :3), n+1 :2) :1)
```

13. Functions to create a descriptor from a list of ranges

These functions are used to construct a descriptor from a list of ranges. This is useful in the construction of any family: e.g., graph objects (nodes or included graphs), ports (graph ports, node ports), and tokens.

```
makeDescriptor (1: var, Range_list :1)          ::=
        if (1: Range_list = null :1)
        then (1: print (2: var, text (3:  = new GCL_Descriptor (); :3) :2) :1)
        else (1: print (2: text (3: { :3) :2)
                declareDescriptorVars (2: listSize (3: Range_list :3) :2)
                makeDescriptorRecurs (2: Range_list, listSize
                                        (3: Range_list :3) :2)
                print (2: var, text (3: = :3),
                        cat (3: text (4: descriptor :4), listSize (4: Range_list :4) :3),
                        text (3: ; :3) :2)
                print (2: text (3: } :3) :2) :1)

declareDescriptorVars (1: n :1)                 ::=
        if (1: n != 0 :1)
        then (1: print (2: text (3: GCL_Descriptor * :3),
                        cat (3: text (4: descriptor :4), n :3), text (3: ; :3),
                        text (3: deque <GCL_Descriptor *> :3),
                        cat (3: text (4: deque :4), n :3), text (3: ; :3) :2)
                declareDescriptorVars (2: n-1 :2) :1)

makeDescriptorRecurs (1: Range_list, n :1)          ::=
        if (1: cdr (2: Range_list :2) = null :1)
        then (1: print (2: cat (3: text (4: descriptor :4), n :3),
                        text (3: = new GCL_Descriptor ( :3),
                        (3: car (4: Range_list :4) :3).Expr_1, text (3: , :3),
                        text (3: ( :3), (3: car (4: Range_list :4) :3).Expr_2,
                        text (3: ) - :3),
                        text (3: ( :3), (3: car (4: Range_list :4) :3).Expr_1,
                        text (3: ) + 1); :3) :2) :1)
        else (1: startLoop (2: car (3: Range_list :3) :2)
                makeDescriptorRecurs (2: cdr (3: Range_list :3), n-1 :2)
                print (2: cat (3: text (4: deque :4), n :3),
                        text (3: . push_back ( :3),
                        cat (3: text (4: descriptor :4), n-1 :3), text (3: ); :3) :2)
                endLoop (2:  :2)
                print (2: cat (3: text (4: descriptor :4), n :3),
                        text (3: = new GCL_Descriptor ( :3),
                        cat (3: text (4: deque  :4), n :3), text (3: , :3),
                        (3: n–1 :3), text (3: , :3),
                        (3: car  (4: Range_list :4) :3).Expr_1, text (3: ); :3),
                        text (3: clearDeque ( :3),
                        cat (3: text (4: deque :4), n :3), text (3: ); :3) :2) :1)
```

14. Functions to build a token

```
makeToken (1: Name, Value, Mode :1)        ::=
    if (1: Value (is a range-leaf) :1)
    then (1: makeTokenFromRangeLeaf (2: Name, Value, Mode :2) :1)
    else (1: makeTokenFromNestedString
        (2: Name, Value.NestedString, Mode :2) :1)
```

The following functions are used to build a token from a list of ranges and a leaf expression.

```
makeTokenFromRangeLeaf (1: Name, Value, Mode :1)          ::=
    makeDescriptor (1: text (2: _descriptor :2), Value.Range_list :1)
    tokenType (1: Mode.BaseType :1)
    print (1: text (2: * :2), Name, text (2: = new :2) :1)
    tokenType (1: Mode.BaseType :1)
    print (1: text (2: (_descriptor); :2),
        Mode.BaseType, text (2: * tokenValue = :2), Name,
        text (2: -> getLeafArray (); :2) :1)
    assignValues (1: Value :1)

tokenType (1: BaseType :1)      ::=
    print (1: text (2: GCL_Token_T < :2), BaseType text (2: > :2) :1)
```

```
assignValues (1: Value :1)                ::=
        print (1: text (2: { unsigned int leafIndex = 0; :2) :1)
        assignEachValue (1: Value.leaf.Expr, Value.Range_list :1)
        print (: text (2: } :2) :1)

assignEachValue (1: Expr, RangeList :1)   ::=
        if (1: RangeList = null :1)
        then (1: print (2: text (3: tokenValue [leafIndex++] = :3),
                        Expr, text (3: ; :3) :2) :1)
        else (1: startLoop (2: car (3: RangeList :3) :2)
                assignEachValue (2: Expr, cdr (3: RangeList :3) :2)
                endLoop (2:  :2) :1)
```

The following functions are used to build a token from a nested string. We assume that the nested string is properly formed. By that, we mean that every leaf in the nested string has the depth specified by the **Mode**.TokHt. This cannot be checked by the GUI or translator. If not, then

a) an error will occur in the execution of one of the subordinate functions,
b) an error will occur during compile time, or
c) an error will occur during run time.

```
makeTokenFromNestedString (1: Name, NestedString, Mode :1)      ::=
        makeDescriptorFromNestedString (1: NestedString,
                        getHeightOfNestedString (2: NestedString :2) :1)
        createBlankToken (1: Name, Mode.BaseType :1)
        setValuesFromNestedString (1: Name, NestedString, Mode :1)

makeDescriptorFromNestedString (1: NestedString, height :1)          ::=
        print (1: { :1)
        makeDescrVars (1: height :1)
        descrFromNS (1: NestedString, height :1)
        print (1: text (2: _descriptor = :2),
                cat (2: text (3: descriptor :3), height :2),
                text (2: ; :2) :1)
        print (1: } :1)
```

Note: The following function examines the nested string to determine the family height. It does so by finding the level of nesting of braces in which the leaf of deepest nesting occurs. We assume that all leaves have the same level of nesting. Otherwise an error will occur as noted above. If there are no leaves, then only the minimum height can be determined from the nested string. In this case, the minimum height is returned.

```
getHeightOfNestedString (1: NestedString :1)      ::=
        if (1: NestedString = null :1) then (1: return (2: 1 :2) :1)
        else if (1: car (2: NestedString :2) = (an expression) :1)
                then (1: return (2: 1 :2) :1)
        else (1: return (2: maximum (3:
                1 + getHeightOfNestedString (4: car (5: NestedString :5) :4),
                getHeightOfNestedString
                        (4: cdr (5: NestedString :5) :4) :3) :2) :1)

makeDescrVars (1: n :1) ::=
        if (1: n >= 0 :1)
        then (1: print (2: text (3: deque <GCL_Descriptor *> :3),
                cat (3: text (4: deque :4), n :3), text (3: ; :3),
                text (3: GCL_Descriptor * :3),
                cat (3: text (4: descriptor :4), n :3), text (3: ; :3) :2)
                makeDescrVars (2: n-1 :2) :1)

descrFromNS (1: NestedString, height :1)  ::=
        if (1: height = 1 :1)
        then (1: print (2: text (3: descriptor1 = new GCL_Descriptor (0, :3),
                        listSize (3: NestedString :3), text (3: ); :3) :2) :1)
        else if (1: NestedString = null :1)
        then (1: print (2: cat (3: text (4: descriptor :4), height :3),
                        text (3: = new GCL_Descriptor (  :3),
                        cat (3: text (4: deque :4), height :3),
                        text (3: , :3), height - 1, text (3: ); :3),
                        text (3: clearDeque ( :3),
                        cat (3: text (4: deque :4) height :3),
                        text (3: ); :3) :2) :1)
        else (1: descrFromNS (2: car (3: NestedString :3), height - 1 :2)
                print (2: cat (3: text (4: deque :4), height :3),
                        text (3: . push_back ( :3),
                        cat (3: text (4: descriptor :4), height - 1 :3),
                        text (3: ); :3) :2)
                descrFromNS (2: cdr (3: NestedString :3), height :2) :1)

createBlankToken (1: Name, BaseType :1) ::=
        print (1: text (2: GCL_Token_T < :2), BaseType, text (2: > * :2),
                Name, text (2: = new GCL_Token_T < :2),
                BaseType, text (2: > ( _descriptor ); :2) :1)
```

```
setValuesFromNestedString (1: Name, NestedString, Mode :1)          ::=
        print (1: Mode.BaseType,
                text (2: leafArray * = :2), Name, text (2: -> getLeafArray (); :2),
                text (2: int leafIndex = 0; :2) :1)
        varsFromNS (1: NestedString :1)

varsFromNS (1: NestedString :1)                  ::=
        if (1: NestedString = null :1) then (1: null :1)
        else if (1: car (2: NestedString :2) (is an expression) :1)
        then (1: print (2: text (3: leafArray [ leafIndex++ ] = :3),
                                car (3: NestedString :3), text (3: ; :3) :2)
                        varsFromNS (2: cdr (3: NestedString :3) :2) :1)
        else (1: varsFromNS (2: car (3: NestedString :3) :2)
                varsFromNS (2: cdr (3: NestedString :3) :2) :1)
```

Appendix A - GUI requirements for Special Transitions

The special transitions Pack and Unpack require special treatment by the GUI. Their class definitions are provided by the Graph Class Library. Therefore no class definitions for them should be provided by the translator.

However there should be "NodeProto" entries in every graph state file for the special transitions. Whenever the user wishes to create a Pack or Unpack transition in a graph, the user indicates this by entering the respective keyword in the "Class Name" field of the transition's call form. If the user wishes to see the prototype form for a Pack or Unpack transition, he can do so. The prototype form for Pack and Unpack must be "read only".

Following are the "NodeProto" specifications for Pack and Unpack (in courier font):

```
transition { prototype Pack fmly (< height, base_type >);
        inport
        {
                READAMOUNT  < 0, unsigned int > ;
                READOFFSET  < 0, unsigned int > ;
                CONSUMEAMOUNT  < 0, unsigned int > ;
                CONSUMEOFFSET  < 0, unsigned int > ;
                INPUT  < height, base_type > ;
        }
        outport
        {
                OUTPUT  < height + 1, base_type > ;
        }
}
```

```
transition { prototype Unpack fmly (< height, base_type >);
        inport
        {
                INPUT  < height + 1, base_type > ;
        }
        outport
        {
                OUTPUT  < height, base_type > ;
                PRODUCE  < 0, unsigned int > ;
        }
}
```

Note: The transition statements in the prototypes of Pack and Unpack are blank, because it is not possible to represent them in the specification of an ordinary transition. They will be provided in the GCL class definitions for Pack and Unpack.

In a similar way there should be "NodeProto" entries in every graph state file for simple queues and graph variables:

```
queue { prototype Queue fmly (< height, base_type >);
        inport
        {
                INPUT  < height, base_type > ;
        }
        outport
        {
                OUTPUT  < height, base_type > ;
        }
}

gvar { prototype GVar fmly (< height, base_type >);
        inport
        {
                INPUT  < height, base_type > ;
        }
        outport
        {
                OUTPUT  < height, base_type > ;
        }
}
```

Notwithstanding the fact that we provide these queue and graph variable prototypes are in every graph state file, the translator specification assumes that these prototypes are not present in the graph state file. A future version of the translator specification will provide functions based on the presence of these prototypes.